2
BACKGROUND AND STATE OF THE ART

This chapter discusses the state-of-the-art in the areas of WebAssembly and Software Diversification. In Section 2.1 we discuss the WebAssembly format, its design and its security model. Besides, we discuss the current state of the art in the area of WebAssembly program analysis. In Section 2.2 we discuss the state of the art in the area of Software Diversification. Morover, we delve into the open challenges regarding the diversification of WebAssembly programs.

2.1 WebAssembly

The W3C publicly announced the WebAssembly (Wasm) language in 2017 as the fourth scripting language supported in all major web browser vendors. Wasm is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al. [**?**] in 2017 and extended by Rossberg et al. in 2018 [**?**]. It is designed to be fast, portable, self-contained and secure, and it outperforms JavaScript execution.

Moreover, WebAssembly has been evolving outside web browsers since its first announcement. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup and memory usage than containerization and virtualization [**?** **?**]. Consequently, in 2019, the Bytecodealliance proposed WebAssembly System Interface (WASI) [**?**]. WASI pioneered the execution of Wasm with a POSIX system interface protocol, making it possible to execute Wasm closer to the underlying operating system. Therefore, it standardizes the adoption of Wasm in heterogeneous platforms [**?**], i.e., IoT and Edge computing [**?** **?**].

Currently, WebAssembly serves a variety of functions in browsers, ranging from gaming to cryptomining [**?**]. Other applications include text processing,

[0]Comp. time 2023/10/16 09:59:56

visualization, media processing, programming language testing, online gambling, bar code and QR code fast reading, hashing, and PDF viewing. On the backend, WebAssembly notably excels in short-running tasks. As such, it is particularly suitable for Function as a Service (FaaS) platforms like Cloudflare and Fastly. The broad spectrum of applicability and the rapid adoption of WebAssembly have resulted in demands for additional features. However, not all these demands align with its original specification. Thus, since the introduction of WebAssembly, various extensions have been proposed for standardization. For instance, the SIMD proposal enables the execution of vectorized instructions in WebAssembly. To become a standard, a proposal must fulfill certain criteria, including having a formal specification and at least two independent implementations, e.g., two different engines. Notably, even after adoption, new extensions are optional; the core WebAssembly remains untouched and continues to be referred to as version 1.0. The subsequent text in this chapter focuses specifically on WebAssembly version 1.0. However, the tools, techniques, and methodologies discussed are applicable to future WebAssembly versions.

2.1.1 From source code to WebAssembly

WebAssembly programs are compiled from source languages like C/C++, Rust, or Go, which means that it can benefit from the optimizations of the source language compiler. The resulting Wasm program is like a traditional shared library, containing instruction codes, symbols, and exported functions. A host environment is in charge of complementing the Wasm program, such as providing external functions required for execution within the host engine. For instance, functions for interacting with an HTML page's DOM are imported into the Wasm binary when invoked from JavaScript code in the browser.

```
1  // Some raw data
2  const int A[250];
3
4  // Imported function
5  int ftoi(float a);
6
7  int main() {
8      for(int i = 0; i < 250; i++) {
9          if (A[i] > 100)
10             return A[i] + ftoi(12.54);
11     }
12
13     return A[0];
14 }
```

Listing 2.1: *Example C program which includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses.*

In Listing 2.1 and Listing 2.2, we present a C program alongside its corresponding WebAssembly binary. The C function encompasses various

elements such as heap allocation, external function usage, and a function definition that includes a loop, conditional branching, function calls, and memory accesses. The Wasm code shown in Listing 2.2 is displayed in its textual format, known as WAT[1]. The static memory declared in line 2 of Listing 2.1 is allocated within the Wasm binary's linear memory, as illustrated in line 47 of Listing 2.2. The function declared in line 5 of Listing 2.1 is converted into an imported function, as seen in line 8 of Listing 2.2. The main function, spanning lines 7 to 14 in Listing 2.1, is transcribed into a Wasm function covering lines 12 to 38 in Listing 2.2. Within this function, the translation of various C language constructs into Wasm can be observed. For instance, the `for` loop found in line 8 of Listing 2.1 is mapped to a block structure in lines 17 to 31 of Listing 2.2. The loop's breaking condition is converted into a conditional branch, as shown in line 25 of Listing 2.2.

[1]The WAT text format is primarily designed for human readability and for low-level manual editing.

```
1  ; WebAssembly magic bytes(\0asm) and version (1.0) ;
2  (module
3  ; Type section: 0x01 0x00 0x00 0x00 0x13 ... ;
4    (type (;0;) (func (param f32) (result i32)))
5    (type (;1;) (func))
6    (type (;2;) (func (result i32)))
7  ; Import section: 0x02  0x00 0x00 0x00 0x57 ... ;
8    (import "env" "ftoi" (func $ftoi (type 0)))
9  ; Custom section: 0x00 0x00 0x00 0x00 0x7E ;
10   (@custom "name" "...")
11 ; Code section: 0x03 0x00 0x00 0x00 0x5B... ;
12   (func $main (type 2) (result i32)
13     (local i32 i32)
14     i32.const -1000
15     local.set 0
16     block  ;label = @1;
17       loop  ;label = @2;
18         i32.const 0
19         local.get 0
20         i32.add
21         i32.load
22         local.tee 1
23         i32.const 101
24         i32.ge_s
25         br_if 1 ;@1;
26         local.get 0
27         i32.const 4
28         i32.add
29         local.tee 0
30         br_if 0 ;@2;
31       end
32       i32.const 0
33       return
34     end
35     f32.const 0x1.9147aep+3
36     call $ftoi
37     local.get 1
38     i32.add)
39 ; Memory section: 0x05 0x00 0x00 0x00 0x03 ... ;
40   (memory (;0;) 1)
41 ; Global section: 0x06 0x00 0x00 0x00 0x11.. ;
42   (global (;4;) i32 (i32.const 1000))
43 ; Export section: 0x07 0x00 0x00 0x00 0x72 ... ;
44   (export "memory" (memory 0))
45   (export "A" (global 2))
46 ; Data section: 0x0d 0x00 0x00 0x03 0xEF  ... ;
47   (data $data (0) "\00\00\00\00...")
48 ; Custom section: 0x00 0x00 0x00 0x00 0x2F ;
49   (@custom "producers" "...")
50 )
```

Listing 2.2: *Wasm code for Listing 2.1. The example Wasm code illustrates the translation from C to Wasm in which several high-level language features are translated into multiple Wasm instructions.*

There exist several compilers that turn source code into WebAssembly binaries. For example, LLVM has offered WebAssembly as a backend option since its 7.1.0 release[2], supporting a diverse set of frontend languages like C/C++,

[2] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`

Rust, Go, and AssemblyScript[3]. Significantly, a study by Hilbig et al. reveals that 70% of WebAssembly binaries are generated using LLVM-based compilers. The main advantage of using LLVM is that it provides a common optimization infrastructure for WebAssembly binaries. In parallel developments, the KMM framework[4] has incorporated WebAssembly as a compilation target.

A recent trend in the WebAssembly ecosystem involves porting various programming languages by converting both the language's engine or interpreter and the source code into a WebAssembly program. For example, Javy[5] encapsulates JavaScript code within the QuickJS interpreter, demonstrating that direct source code conversion to WebAssembly isn't always required. If an interpreter for a specific language can be compiled to WebAssembly, it allows for the bundling of both the interpreter and the language into a single, isolated WebAssembly binary. Similarly, Blazor[6] facilitates the execution of .NET Common Intermediate Language (CIL) in WebAssembly binaries for browser-based applications. However, this approach is still non-mature and faces challenges, such as the absence of JIT compilation for the "interpreted" code, making it less suitable for long-running tasks [**?**]. On the other hand, it proves effective for short-running tasks, particularly those executed in Edge-Cloud computing platforms.

2.1.2 WebAssembly's binary format

The Wasm binary format is close to machine code and already optimized, being a consecutive collection of sections. In Figure 2.1 we show the binary format of a Wasm section. A Wasm section starts with a 1-byte section ID, followed by a 4-byte section size, and concludes with the section content, which precisely matches the size indicated earlier. A Wasm binary contains sections of 13 types, each with a specific semantic role and placement within the module. For instance, the *Custom Section* stores metadata like the compiler used to generate the binary, while the *Type Section* contains function signatures that serve to validate the *Function Section*. The *Import Section* lists elements imported from the host, and the *Function Section* details the functions defined within the binary. Other sections like *Table, Memory, and Global Sections* specify the structure for indirect calls, unmanaged linear memories, and global variables, respectively. *Export, Start, Element, Code, Data, and Data Count Sections* handle aspects ranging from declaring elements for host engine access to initializing program state, declaring bytecode instructions per function, and initializing linear memory. Each of these sections must occur only once in a binary and can be empty, i.e., they can

[3] A subset of the TypeScript language
[4] `https://kotlinlang.org/docs/wasm-overview.html`
[5] `https://github.com/bytecodealliance/javy`
[6] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`

be empty. For the sake of understanding, we also annotate sections as comments in the Wasm code in Listing 2.2.

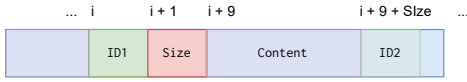


Figure 2.1: *Memory byte representation of a WebAssembly binary section, starting with a 1-byte section ID, followed by an 8-byte section size, and finally the section content.*

A Wasm binary can be processed efficiently due to its organization into a contiguous array of sections. For instance, this structure permits compilers to expedite the compilation process either through parallel parsing or by disregarding *Custom Sections*. Moreover, the *Code Section*'s instructions are further compacted through the use of the LEB128[7] encoding. Consequently, Wasm binaries are not only fast to validate and compile, but also swift to transmit over a network.

2.1.3 WebAssembly's runtime

The WebAssembly's runtime characterizes the behavior of Wasm programs during execution. This section describes the main components of the WebAssembly runtime, namely the execution stack, functions, memory model, and execution process. These components are crucial for understanding both the WebAssembly's control flow and the analysis of WebAssembly binaries.

Execution Stack: At runtime, WebAssembly engines instantiate a WebAssembly module. This module is a runtime representation of a loaded and initialized WebAssembly binary described in Subsection 2.1.2. The primary component of a module instance is its Execution Stack. The Execution Stack stores typed values, labels, and control frames. Labels manage block instructions, loops, and function calls. Values within the stack can only be static types. These types include `i32` for 32-bit signed integers, `i64` for 64-bit signed integers, `f32` for 32-bit floats, and `f64` for 64-bit floats. Abstract types such as classes, objects, and arrays are not supported natively. Instead, these types are abstracted into primitive types during compilation and stored in linear memory.

Functions: At runtime, WebAssembly functions are closures over the module instance, grouping locals and function bodies. Locals are typed variables that are local to a specific function invocation. A function body is a sequence of instructions that are executed when the function is called. Each instruction either reads from the execution stack, writes to the execution stack, or modifies the control flow of the function. Recalling the example Wasm binary previously

[7] `https://en.wikipedia.org/wiki/LEB128`

showed, the local variable declarations and typed instructions that are evaluated using the stack can be appreciated between Line 7 and Line 32 in Listing 2.2. Each instruction reads its operands from the stack and pushes back the result. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add`. As the listing also shows, instructions are annotated with a numeric type.

Memory model: A WebAssembly module instance incorporates three key types of memory-related components. These components can either be managed solely by the host engine or shared with the WebAssembly binary itself. This division of responsibility is often categorized as *managed* and *unmanaged* memory [**?**]. Managed refers to components that are exclusively modified by the host engine at the lowest level, e.g. when the WebAssembly binary is JITed, while unmanaged components can also be altered through WebAssembly opcodes. First, modules may include multiple linear memory instances, which are contiguous arrays of bytes. These are accessed using 32-bit integers (`i32`) and are shareable only between the initiating engine and the WebAssembly binary. Generally, these linear memories are considered to be unmanaged, e.g. line 21 of Listing 2.2 shows an explicit memory opcode. Second, there are global instances, which are variables accompanied by values and mutability flags (see example in line 42 of Listing 2.2). These globals are managed by the host engine, which controls their allocation and memory placement completely oblivious to the WebAssembly binary scope. They can only be accessed via their declaration index, prohibiting dynamic addressing. Third, local variables are mutable and specific to a given function instance. They are accessible only through their index relative to the executing function and are part of the data managed by the host engine.

WebAssembly module execution: WebAssembly is optimized and closely aligned with machine code for performance reasons, which typically results in fast JIT compilation for execution. Engines such as V8[8] and SpiderMonkey[9] employ this strategy when executing WebAssembly binaries in browser clients. After JITed, the WebAssembly binary operates within a sandboxed environment, accessing the host environment only via imported JavaScript functions from the host. While WebAssembly was initially developed for browsers, it has significantly evolved, primarily due to WASI[**?**]. WASI provides a standardized POSIX-like interface for interactions between WebAssembly modules and host environments. Compilers can generate Wasm binaries that employ WASI, enabling execution in standalone engines. These binaries can subsequently be executed by standalone engines across various environments, including the cloud, servers, and IoT devices. Much like browsers, these engines often convert WebAssembly into machine code via JIT compilation, ensuring a sandboxed execution process. Standalone engines

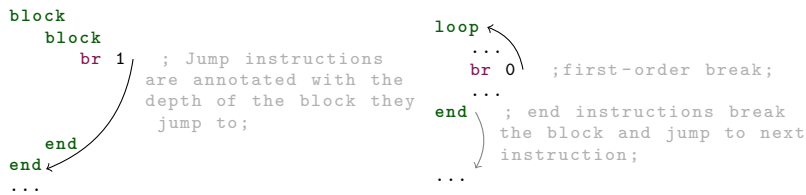
[8] `https://chromium.googlesource.com/v8/v8.git`
[9] `https://spidermonkey.dev/`

such as WASM3[10], Wasmer[11], Wasmtime[12], WAVM[13], and Sledge[**?**] have been developed to support both WebAssembly and WASI. In a similar vein, Singh et al.[**?**] created a WebAssembly virtual machine explicitly designed for Arduino-based devices.

2.1.4 WebAssembly's control flow

A WebAssembly function groups instructions which are organized into blocks, with the function's entrypoint point acting as the root block. In contrast to conventional assembly code, control flow structures in Wasm leap between block boundaries rather than arbitrary positions within the code, effectively prohibiting `goto`s to random code positions. Each block may specify the needed execution stack state before execution as well as the resultant execution stack state once its instructions have been executed. Typically, the execution stack state is simply the quantity and numeric type of values on the stack. This stack state is used to validate the binary during compilation and to ensure that the stack is in a valid state before the execution of the block's instructions. Blocks in Wasm are explicit (see instructions `block` and `end` in lines 16 and 34 of Listing 2.2), delineating where they commence and conclude. By design, a block cannot reference or execute code from external blocks.

During runtime, WebAssembly break instructions can only jump to one of its enclosing blocks. Breaks, except for those within loop constructions, jump to the block's end and continue to the next immediate instruction. For instance, after line 34 of Listing 2.2, the execution would proceed to line 35. Within a loop, the end of a block results in a jump to the block's beginning, thus restarting the loop. For example, if line 30 of Listing 2.2 evaluates as false, the next instruction to be executed in the loop would be line 18. Listing 2.3 provides an example for better understanding, comparing a standard block and a loop block in a Wasm function.



Listing 2.3: *Example of breaking a block and a loop in WebAssembly.*

[10] https://github.com/wasm3/wasm3
[11] https://wasmer.io/
[12] https://github.com/bytecodealliance/wasmtime
[13] https://github.com/WAVM/WAVM

Each break instruction includes the depth of the enclosing block as an operand. This depth is used to identify the target block for the break instruction. For example, in the left-most part of the previously discussed listing, a break instruction with a depth of 1 would jump past two enclosing blocks. This design hardens the rewriting of Wasm binaries. For instance, if a block is removed, the depth of the break instructions within the block must be updated to reflect the new enclosing block depth. This is a significant challenge for rewriting tools, as it requires the analysis of the control flow graph to determine the enclosing block depth for each break instruction.

2.1.5 Security and Reliability for WebAssembly

The WebAssembly ecosystem's expansion needs robust tools to ensure its security and reliability. Numerous tools, employing various strategies to detect vulnerabilities in WebAssembly programs, have been created to meet this need. This paper presents a review of the most relevant tools in this field, focusing on those capable of providing security guarantees for WebAssembly binaries.

Static analysis: SecWasm[**?**] uses information control flow strategies to identify vulnerabilities in WebAssembly binaries. Conversely, Wasmati[**?**] employs code property graphs for this purpose. Wasp[**?**] leverages concolic execution to identify potential vulnerabilities in WebAssembly binaries. VeriWasm[**?**], an offline verifier designed specifically for native x86-64 binaries JITed from WebAssembly, adopts a unique approach. While these tools emphasize specific strategies, others adopt a more holistic approach. CT-Wasm[**?**], verifies the implementation of cryptographic algorithms in WebAssembly. For example, both Wassail[**?**] and WasmA[**?**] provide a comprehensive static analysis framework for WebAssembly binaries. However, static analysis tools may have limitations. They may fail to detect vulnerabilities triggered at runtime by a specific WebAssembly input. For instance, a newly, semantically equivalent WebAssembly binary may be generated from the same source code bypassing or breaking the static analysis [**?**]. If the WebAssembly input differs from the input used during static analysis, the vulnerability may go unnoticed. Thus, there may be a lack of subjects to evaluate the effectiveness of these tools.

Dynamic analysis: Dynamic analysis involves tools such as TaintAssembly[**?**], which conducts taint analysis on WebAssembly binaries. Fuzzm[**?**] identifies vulnerabilities in host engines by conducting property fuzzing through WebAssembly binary execution. Furthermore, Stiévenart and colleagues have developed a dynamic approach to slicing WebAssembly programs based on Observational-Based Slicing (ORBS)[**?** **?**]. Similarly, Vivienne applies relational Symbolic Execution (SE) to WebAssembly binaries in order to reveal vulnerabilities in cryptographic implementations[**?**]. This technique aids in debugging, understanding programs, and conducting security analysis. However,

Wasabi[**?**] remains the only general-purpose dynamic analysis tool for WebAssembly binaries, primarily used for profiling, instrumenting, and debugging WebAssembly code. Similar to static analysis, these tools typically analyze software behavior during execution, making them inherently reactive. In other words, they can only identify vulnerabilities or performance issues while pseudo-executing input Wasm programs. Thus, facing an important limitation on overhead for real-world scenarios.

Protecting WebAssembly binaries and runtimes: The techniques discussed previously are primarily focused on reactive analysis of WebAssembly binaries. However, there exist approaches to harden WebAssembly binaries, enhancing their secure execution, and fortifying the security of the entire execution runtimes ecosystem. For instance, Swivel[**?**] proposes a compiler-based strategy designed to eliminate speculative attacks on WebAssembly binaries, particularly in Function-as-a-Service (FaaS) platforms such as Fastly. Conversely, WaVe[**?**] introduces a mechanized engine for WebAssembly that facilitates differential testing. This engine can be employed to detect anomalies in engines running Wasm-WASI programs. Much like static and dynamic analysis tools, these tools may suffer from a lack of WebAssembly inputs, which could affect the measurement of their effectiveness.

WebAssembly malware: The web has seen a steady prevalence of cryptomalware since the inception of Wasm. This is primarily due to the transition of mining algorithms from using CPUs to Wasm, a change driven by clear performance benefits [**?**]. Tools like MineSweeper[**?**], MinerRay[**?**], and MINOS[**?**] use static analysis through machine learning techniques to detect browser-based cryptomalwares. On the other hand, tools like SEISMIC[**?**], RAPID[**?**], and OUTGuard[**?**] utilize dynamic analysis techniques to achieve the same goal. VirusTotal[14], which incorporates over 60 commercial antivirus systems as black-boxes, can detect cryptomalware in Wasm binaries. However, certain obfuscation studies have revealed their flaws. The seminal work of Bahnsali et al. [**?**] demonstrated that previous techniques could be evaded using obfuscation techniques directly on the cryptomining algorithm's source code.

Open challenges: Despite advancements in WebAssembly analysis, several challenges persist. WebAssembly, although deterministic and well-typed by design, has an emerging ecosystem vulnerable to various security threats. Most notably, as previously noted, most works on WebAssembly sit on the reactive side. Their implementation are based on already reported vulnerabilities. Thus, WebAssembly binaries and runtime implementations might be susceptible to unidentified attacks. Conversely, these techniques may not be tested in real-world scenarios, analyzing a limited set of WebAssembly binaries. Moreover,

[14] `https://www.virustotal.com`

Side-channel attacks pose a significant threat. For example, Genkin et al. demonstrated that WebAssembly can be exploited to extract data through cache timing-side channels [**?**]. Research by Maisuradze and Rossow also showed the possibility of speculative execution attacks on WebAssembly binaries [**?**]. Rokicki et al. revealed the potential for port contention side-channel attacks on WebAssembly binaries in browsers [**?**]. Furthermore, studies by Lehmann et al. and Stiévenart et al. suggested that vulnerabilities in C/C++ source code could propagate into WebAssembly binaries [**?** **?**].

This dissertation presents a comprehensive toolset designed to enhance WebAssembly security proactively through Software Diversification. First, Software Diversification expands the capabilities of the mentioned tools by incorporating diversified program variants, making it more challenging for attackers to exploit any missed vulnerabilities through static and dynamic analysis. Generated as proactive security, these diversified variants can simulate a broader set of real-world conditions, thereby making the analysis more accurate. Second, we noted that current solutions to mitigate side-channel attacks on WebAssembly binaries are either specific to certain attacks or need the modification of runtimes, e.g., Swivel as a cloud-deployed compiler. Software Diversification could mitigate yet-unknown vulnerabilities on WebAssembly binaries by generating diversified variants in a platform-agnostic manner.

2.2 Software diversification

Software Diversification has been widely studied in the past decades. This section discusses its state-of-the-art. Software diversification consists in synthesizing, reusing, distributing, and executing different, functionally equivalent programs. According to the survey by Baudry et al. [**?**], the motivation for software diversification can be separated in five categories: reusability [**?**], software testing [**?**], performance [**?**], fault tolerance [**?**] and security [**?**]. Our work contributes to the latter two categories. In this section we discuss related works by highlighting how they generate diversification and how they put it into practice.

TODO Work on differential testing `https://arxiv.org/pdf/2309.12167.pdf`

2.2.1 Generation of Software Variants

There are two primary sources of software diversification: Natural Diversity and Artificial Diversity[**?**]. This work contributes to the state of the art of Artificial Diversity, which consists of software synthesis. This thesis is founded on the work of Cohen in 1993 [**?**] as follows.

Cohen et al. [**?**] proposed to generate artificial software diversification through mutation strategies. A mutation strategy is a set of rules to define how

a specific component of software development should be changed to provide a different yet functionally equivalent program. Cohen and colleagues proposed 10 concrete transformation strategies that can be applied at fine-grained levels. All described strategies can be mixed together. They can be applied in any sequence and recursively, providing a richer diversity environment. We summarize them, complemented with the work of Baudry et al. [**?**] and the work of Jackson et al. [**?**], in 5 strategies.

Equivalent instructions replacement Semantically equivalent code can replace pieces of programs. This strategy replaces the original code with equivalent arithmetic expressions or injects instructions that do not affect the computation result. There are two main approaches for generating equivalent code: rewriting rules and exhaustive searching. The replacement strategies are written by hand as rewriting rules for the first one. A rewriting rule is a tuple composed of a piece of code and a semantic equivalent replacement. For example, Cleemput et al. [**?**] and Homescu et al. [**?**] insert NOP instructions to generate statically different variants. In their works, the rewriting rule is defined as `instr => (nop instr)`, meaning that `nop` operation followed by the instruction is a valid replacement . On the other hand, exhaustive searching samples all possible programs for a specific language. In this topic, Jacob et al. [**?**] proposed the technique called superdiversification for x86 binaries. The superdiversification strategy proposed by Jacob and colleagues performs an exhaustive search of all programs that can be constructed from a specific language grammar. If one of the generated programs is equivalent to the original program, then it is reported as a variant. Similarly, Tsoupidi et al. [**?**] introduced Diversity by Construction, a constraint-based compiler to generate software diversity for MIPS32 architecture.

Instruction reordering This strategy reorders instructions or entire program blocks if they are independent. The location of variable declarations might change as well if compilers re-order them in the symbol tables. It prevents static examination and analysis of parameters and alters memory locations. In this field, Bhatkar et al. [**?** **?**] proposed the random permutation of the order of variables and routines for ELF binaries.

Adding, changing, removing jumps and calls This strategy creates program variants by adding, changing, or removing jumps and calls in the original program. Cohen [**?**] mainly illustrated the case by inserting bogus jumps in programs. Pettis and Hansen [**?**] proposed to split basic blocks and functions for the PA-RISC architecture, inserting jumps between splits. Similarly, Crane et al. [**?**] de-inline basic blocks of code as an LLVM pass. In their approach, each de-inlined code is transformed into semantically equivalent functions that are randomly selected at runtime to replace the original code calculation. On the same topic, Bhatkar et al. [**?**] extended their previous approach [**?**], replacing function calls by indirect pointer calls in C source code, allowing post binary reordering of function calls. Recently, Romano et al. [**?**] proposed an obfuscation technique for JavaScript in which part of the code is replaced by calls to complementary Wasm function.

Program memory and stack randomization This strategy changes the layout of programs in the host memory. Also, it can randomize how a program variant operates its memory. The work of Bhatkar et al. [**?** **?**] propose to randomize the base addresses of applications and the library memory regions in ELF binaries. Tadesse Aga and Autin [**?**], and Lee et al. [**?**] propose a technique to randomize the local stack organization for function calls using a custom LLVM compiler. Younan et al. [**?**] propose to separate a conventional stack into multiple stacks where each stack contains a particular class of data. On the same topic, Xu et al. [**?**] transforms programs to reduce memory exposure time, improving the time needed for frequent memory address randomization.

ISA randomization and simulation This strategy uses a key to cypher the original program binary into another encoded binary. Once encoded, the program can be decoded only once at the target client, or it can be interpreted in the encoded form using a custom virtual machine implementation. This technique is strong against attacks involving code inspection. Kc et al. [**?**], and Barrantes et al. [**?**] proposed seminal works on instruction-set randomization to create a unique mapping between artificial CPU instructions and real ones. On the same topic, Chew and Song [**?**] target operating system randomization. They randomize the interface between the operating system and the user applications. Couroussé et al. [**?**] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. Their technique generates a different program during execution using an interpreter for their DSL. Code obfuscation [**?**] can be seen as a simplification of *ISA randomization*. The main difference between encoding and obfuscating code is that the former requires the final target to know the encoding key while the latter executes as is in any client. Yet, both strategies are meant to tackle program analysis from potential attackers.

Equivalence checking between program variants is an essential component for any program transformation task, from checking compiler optimizations [**?**] to the artificial synthesis of programs discussed in this chapter. Equivalence checking proves that two pieces of code or programs are semantically equivalent [**?**]. Cohen [**?**] simplifies this checking by enunciating the following property: two programs are equivalent if given identical input, they produce the identical output. We use this same enunciation as the definition of *functional equivalence* along with this dissertation. Equivalence checking in Software Diversification aims to preserve the original functionality for programs while changing observable behaviors. For example, two programs can be statically different or have different execution times and provide the same computation.

The equivalence property is often guaranteed by construction. For example, in the case illustrated in Subsection 2.2.1 for Cleemput et al. [**?**] and Homescu et al. [**?**], their transformation strategies are designed to generate semantically equivalent program variants. However, this process is prone to developer errors, and further validation is needed. For example, the test suite of the original program can be used to check the variant. If the test suite passes for the program

variant [**?**], then this variant can be considered equivalent to the original. However, this technique is limited due to the need for a preexisting test suite. When the test suite does not exist, another technique is needed to check for equivalence.

If there is no test suite or the technique does not inherently implement the equivalence property, the previously mentioned works use theorem solvers (SMT solvers) [**?**] to prove equivalence. For SMT solvers, the main idea is to turn the two code variants into mathematical formulas. The SMT solver checks for counter-examples. When the SMT solver finds a counter-example, there exists an input for which the two mathematical formulas return a different output. The main limitation of this technique is that all algorithms cannot be translated to a mathematical formula, for example, loops. Yet, this technique tends to be the most used for no-branching-programs checking like basic block and peephole replacements [**?**].

Another approach to check equivalence between two programs similar to using SMT solvers is by using fuzzers [**?**]. Fuzzers randomly generate inputs that provide different observable behavior. If two inputs provide a different output in the variant, the variant and the original program are not equivalent. The main limitation for fuzzers is that the process is remarkably time-expensive and requires the introduction of oracles by hand.

Definition 1. *Software variant* **TODO** *Define*

Definition 2. *Uncontrolled diversification TODO*

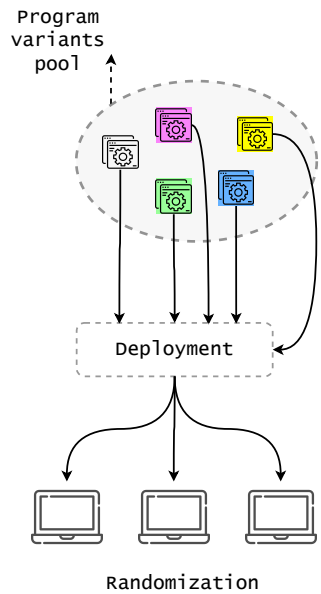
Definition 3. *Controlled diversification TODO*

2.2.2 Variants deployment

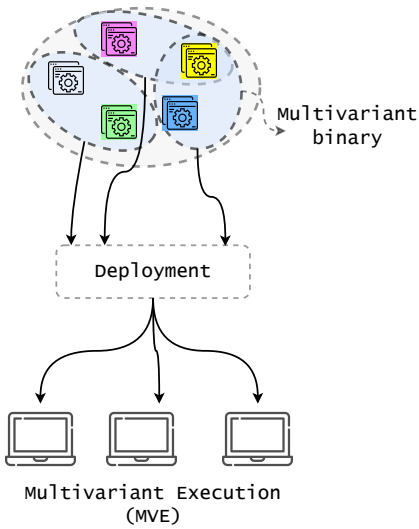
After program variants are generated, they can be used in two main scenarios: Randomization or Multivariant Execution (MVE) [**?**]. In Figure 2.2a and Figure 2.2b we illustrate both scenarios.

Randomization: In the context of our work *Randomization* refers to the ability of a program to be served as different variants to different clients. In the scenario of Figure 2.2a, a program is selected from the collection of variants (program's variant pool), and at each deployment, it is assigned to a random client. Jackson et al. [**?**] compare the variant's pool in Randomization with a herd immunity, since vulnerable binaries can affect only part of the client's community.

El-Khalil and colleagues [**?**] propose to use a custom compiler to generate different binaries out of the compilation process. El-Khalil and colleagues modify a version of GCC 4.1 to separate a conventional stack into several component parts, called multistacks. On the same topic, Aga and colleagues [**?**] propose to generate program variants by randomizing its data layout in memory. Their approach makes each variant to operate the same data in memory with different



(a) Randomization scenario. Given a pool of program variants, one variant is deployed per host. Each deployment randomly selects which variant is assigned to each host. The same program variant is executed in the host at every program invocation between deployments.



(b) Multivariant Execution scenario. Given a pool of program variants, a sample of the pool is packaged in a multivariant binary that is deployed. Each deployment randomly selects which multivariant binary is assigned to each host. Finally, a variant from the multivariant binary is randomly executed at runtime in the host.

Figure 2.2: *Software Diversification usages.*

memory offsets. The Polyverse company[15] materializes randomization at the commercial level in real life. They deliver a unique Linux distribution compilation for each of its clients by scrambling the Linux packages at the source code level.

Virtual machines and operating systems can be also randomized. On this topic, Kc et al. [**?**], create a unique mapping between artificial CPU instructions and real ones. Their approach makes possible the assignment of different variants to specific target clients. Similarly, Xu et al. [**?**] recompile the Linux Kernel to reduce the exposure time of persistent memory objects, increasing the frequency of address randomization.

Multivariant Execution (MVE): Multiple program variants are composed in one single binary (multivariant binary) [**?**]. Each multivariant binary is randomly deployed to a client. Once in the client, the multivariant binary executes its embedded program variants at runtime. Figure 2.2b illustrates this scenario.

[15] `https://polyverse.com/`

The execution of the embedded variants can be either in parallel to check for inconsistencies or a single program to randomize execution paths [**?**]. Bruschi et al. [**?**] extended the idea of executing two variants in parallel with not-overlapping and randomized memory layouts. Simultaneously, Salamat et al. [**?**] modified a standard library that generates 32-bit Intel variants where the stack grows in the opposite direction, checking for memory inconsistencies. Notably, Davi et al. proposed Isomeron [**?**], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. The previously mentioned works showed the benefits of exploiting the limit case of executing only two variants in a multivariant environment. Agosta et al. [**?**] and Crane et al. [**?**] used more than two generated programs in the multivariant composition, randomizing software control flow at runtime.

Both scenarios have demonstrated to harden security by tackling known vulnerabilities such as (JIT)ROP attacks [**?**] and power side-channels [**?**]. Moreover, Artificial Software Diversification is a preemptive technique for yet unknown vulnerabilities [**?**]. Our work contributes to both usage scenarios for Wasm.

TODO Multivariant

Definition 4. *Multivariant* ***TODO*** *Define*

TODO Automatic, SMT based **TODO** Take a look to Jackson thesis, we have a similar problem he faced with the superoptimization of NaCL **TODO** By design **TODO** Introduce the notion of rewriting rule by Sasnaukas. `https://link.springer.com/chapter/10.1007/978-3-319-68063-7_13`

2.2.3 Defensive Diversification

2.2.4 Offensive Diversification

2.3 Open challenges

In **??** we list the related work on Artificial Software Diversification discussed along with this chapter. The first column in the table correspond to the author names and the references to their work, followed by one column for each strategy and usage (Subsection 2.2.1, Subsection 2.2.1, Subsection 2.2.1, Subsection 2.2.1, Subsection 2.2.1, Figure 2.2.2 and Figure 2.2.2). The last column of the table summarizes the technical contribution and the reach of the referred work. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year

of the work in ascending order. In the following text, we enumerate the open challenges we have found in the literature research:

1. *Software monoculture:* The same Wasm code is executed in millions of clients devices through web browser. In addition, Wasm evolves to support edge-cloud computing platforms in backend scenarios, *i.e.,* replicating the same binary along with all computing nodes in a worldwide scale. Therefore, potential vulnerabilities are spread, highlighting a monoculture phenomenon [**?**].

2. *Lack of Software Diversification for Wasm*: Software Diversification has demonstrated to provide protection for known and yet-unknown vulnerabilities. However, only one software diversity approach has been applied to the context of Wasm [**?**]. Moreover, Wasm is a novel technology and, the adoption of defenses is still under development [**?** **?**] and has a low pace, making software diversification a possible preemptive technique. Besides, the preexisting works based on the LLVM pipeline cannot be extended to Wasm because they contribute to LLVM versions released before the inclusion of Wasm as an architecture.

3. *Lack of research on MVE for Wasm:* Wasm has a growing adoption for Edge platforms. However, MVE in a distributed setting like the Edge has been less researched. Only Voulimeneas et al. [**?**] recently proposed a multivariant execution system by parallelizing the execution of the variants in different machines for the sake of efficiency.